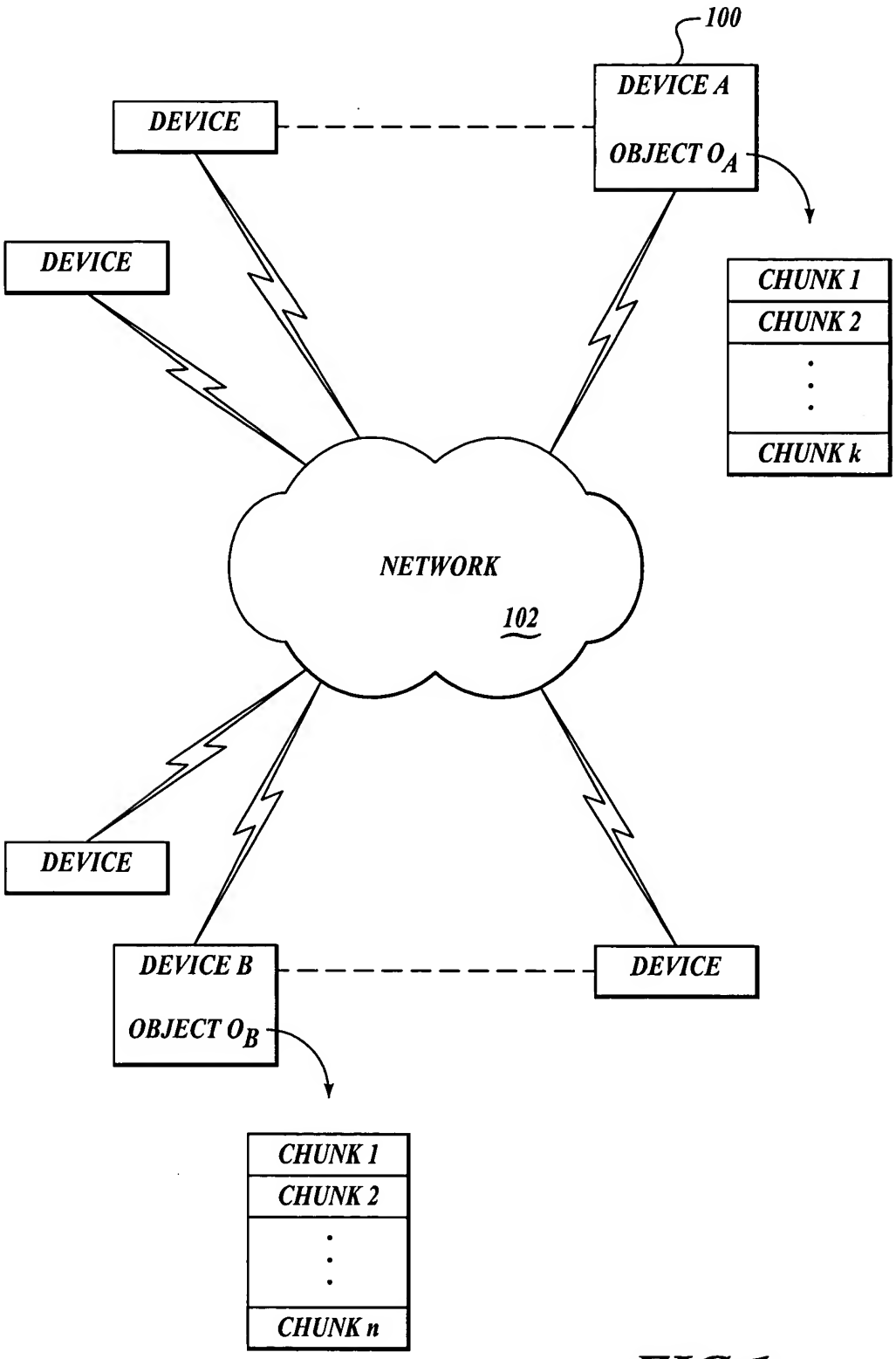

FIG.1

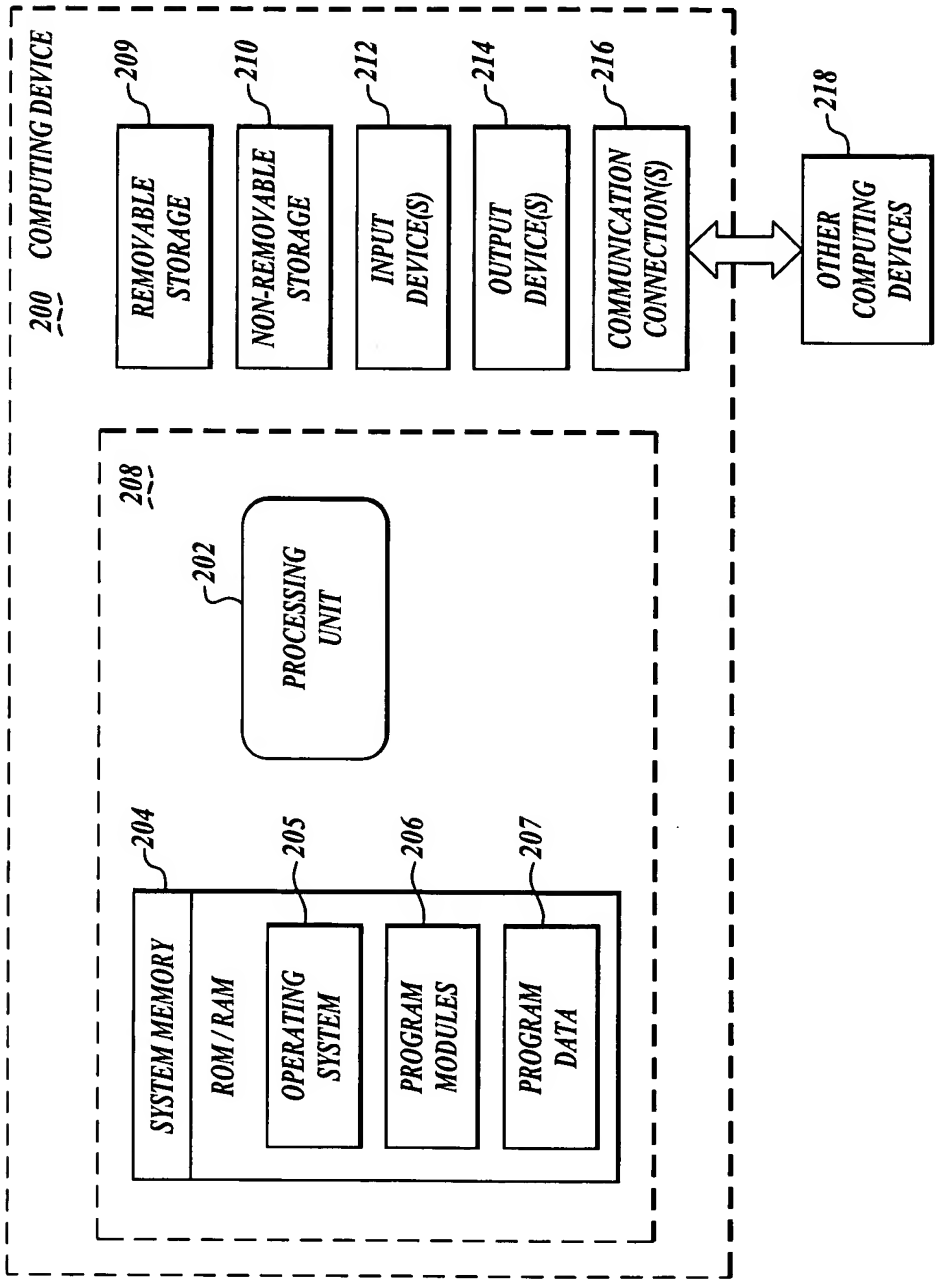
200 COMPUTING DEVICE
209 REMOVABLE STORAGE
210 NON-REMOVABLE STORAGE
212 INPUT DEVICE(S)
214 OUTPUT DEVICE(S)
216 COMMUNICATION CONNECTION(S)
218 OTHER COMPUTING DEVICES
208
202 PROCESSING UNIT
204 SYSTEM MEMORY
ROM / RAM
205 OPERATING SYSTEM
206 PROGRAM MODULES
207 PROGRAM DATA

FIG.2

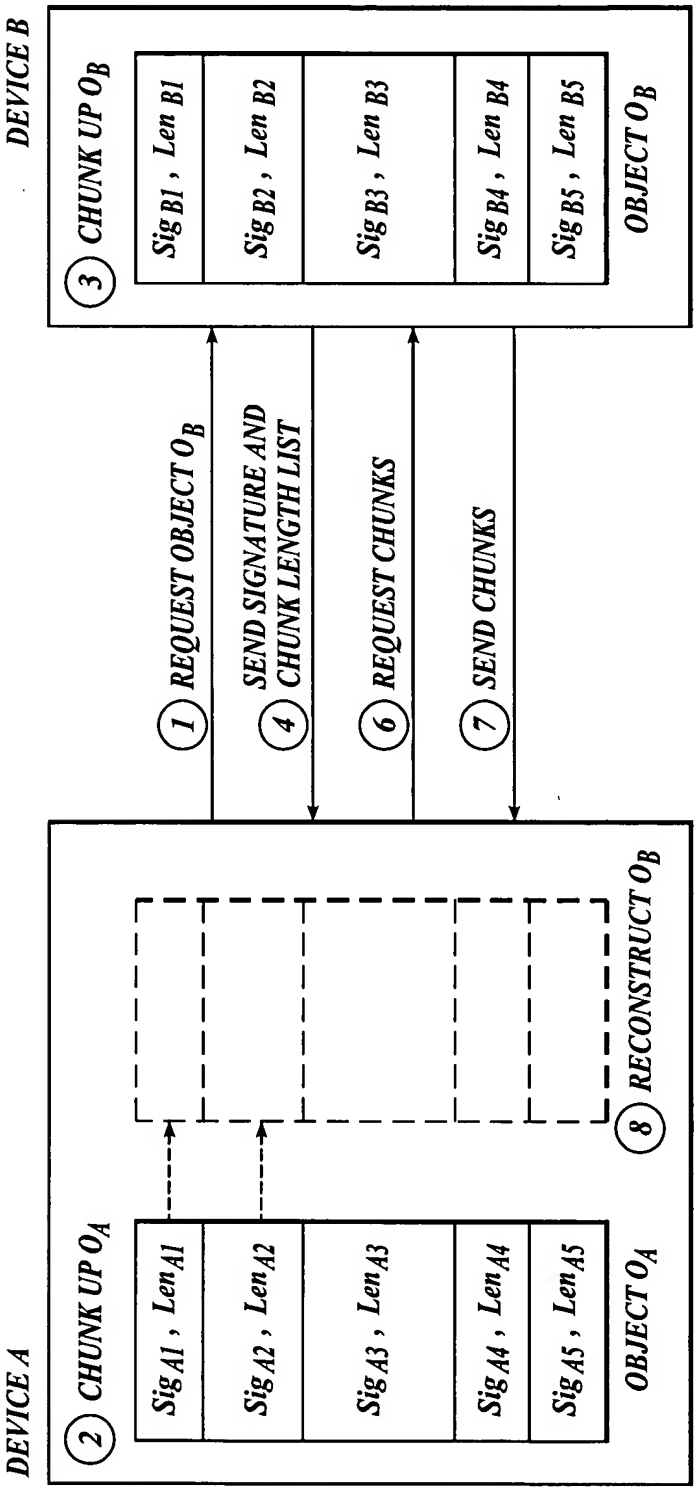
DEVICE B
3 CHUNK UP O_B
Sig B1 , Len B1
Sig B2 , Len B2
Sig B3 , Len B3
Sig B4 , Len B4
Sig B5 , Len B5
OBJECT O_B
1 REQUEST OBJECT O_B
4 SEND SIGNATURE AND CHUNK LENGTH LIST
6 REQUEST CHUNKS
7 SEND CHUNKS
DEVICE A
2 CHUNK UP O_A
Sig A1 , Len A1
Sig A2 , Len A2
Sig A3 , Len A3
Sig A4 , Len A4
Sig A5 , Len A5
OBJECT O_A
8 RECONSTRUCT O_B

FIG.3A

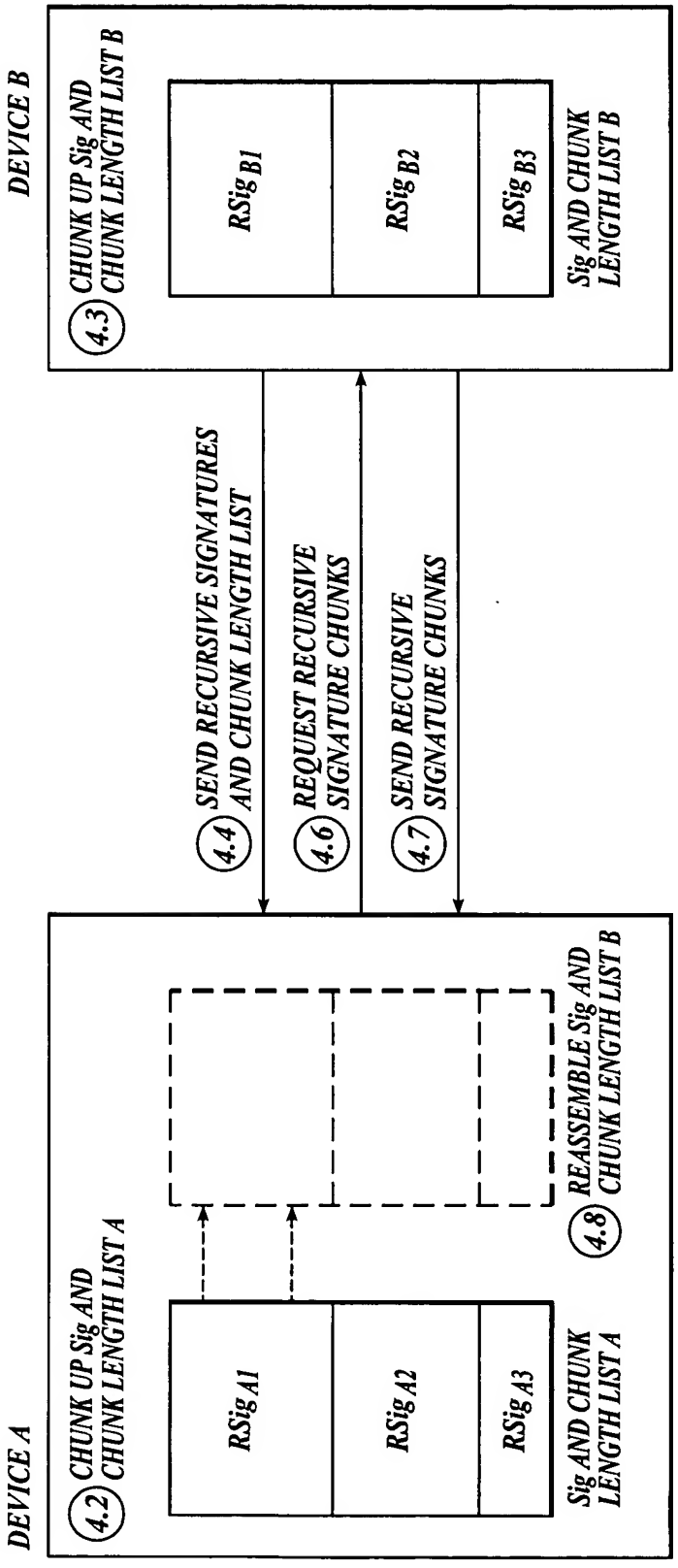
DEVICE B
(4.3) CHUNK UP Sig AND CHUNK LENGTH LIST B
RSig B1
RSig B2
RSig B3
Sig AND CHUNK LENGTH LIST B
(4.4) SEND RECURSIVE SIGNATURES AND CHUNK LENGTH LIST
(4.6) REQUEST RECURSIVE SIGNATURE CHUNKS
(4.7) SEND RECURSIVE SIGNATURE CHUNKS
DEVICE A
(4.2) CHUNK UP Sig AND CHUNK LENGTH LIST A
RSig A1
RSig A2
RSig A3
Sig AND CHUNK LENGTH LIST A
(4.8) REASSEMBLE Sig AND CHUNK LENGTH LIST B

FIG.3B

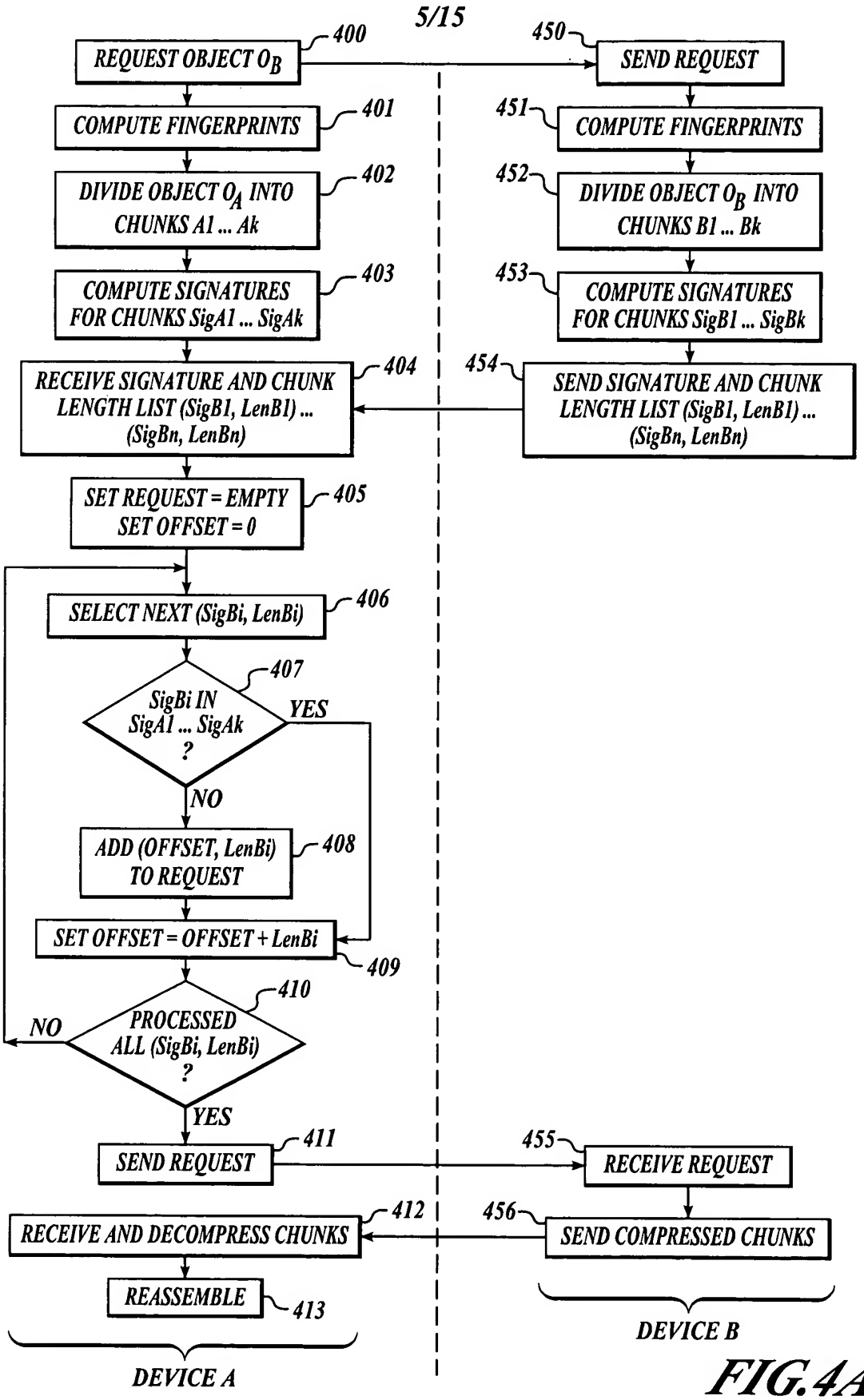
400
REQUEST OBJECT O_B
401
COMPUTE FINGERPRINTS
402
DIVIDE OBJECT O_A INTO CHUNKS A1 ... Ak
403
COMPUTE SIGNATURES FOR CHUNKS SigA1 ... SigAk
404
RECEIVE SIGNATURE AND CHUNK LENGTH LIST (SigB1, LenB1) ... (SigBn, LenBn)
405
SET REQUEST = EMPTY
SET OFFSET = 0
406
SELECT NEXT (SigBi, LenBi)
407
SigBi IN SigA1 ... SigAk ?
YES
NO
408
ADD (OFFSET, LenBi) TO REQUEST
409
SET OFFSET = OFFSET + LenBi
410
PROCESSED ALL (SigBi, LenBi) ?
NO
YES
411
SEND REQUEST
412
RECEIVE AND DECOMPRESS CHUNKS
413
REASSEMBLE
DEVICE A
450
SEND REQUEST
451
COMPUTE FINGERPRINTS
452
DIVIDE OBJECT O_B INTO CHUNKS B1 ... Bk
453
COMPUTE SIGNATURES FOR CHUNKS SigB1 ... SigBk
454
SEND SIGNATURE AND CHUNK LENGTH LIST (SigB1, LenB1) ... (SigBn, LenBn)
455
RECEIVE REQUEST
456
SEND COMPRESSED CHUNKS
DEVICE B

FIG. 4A

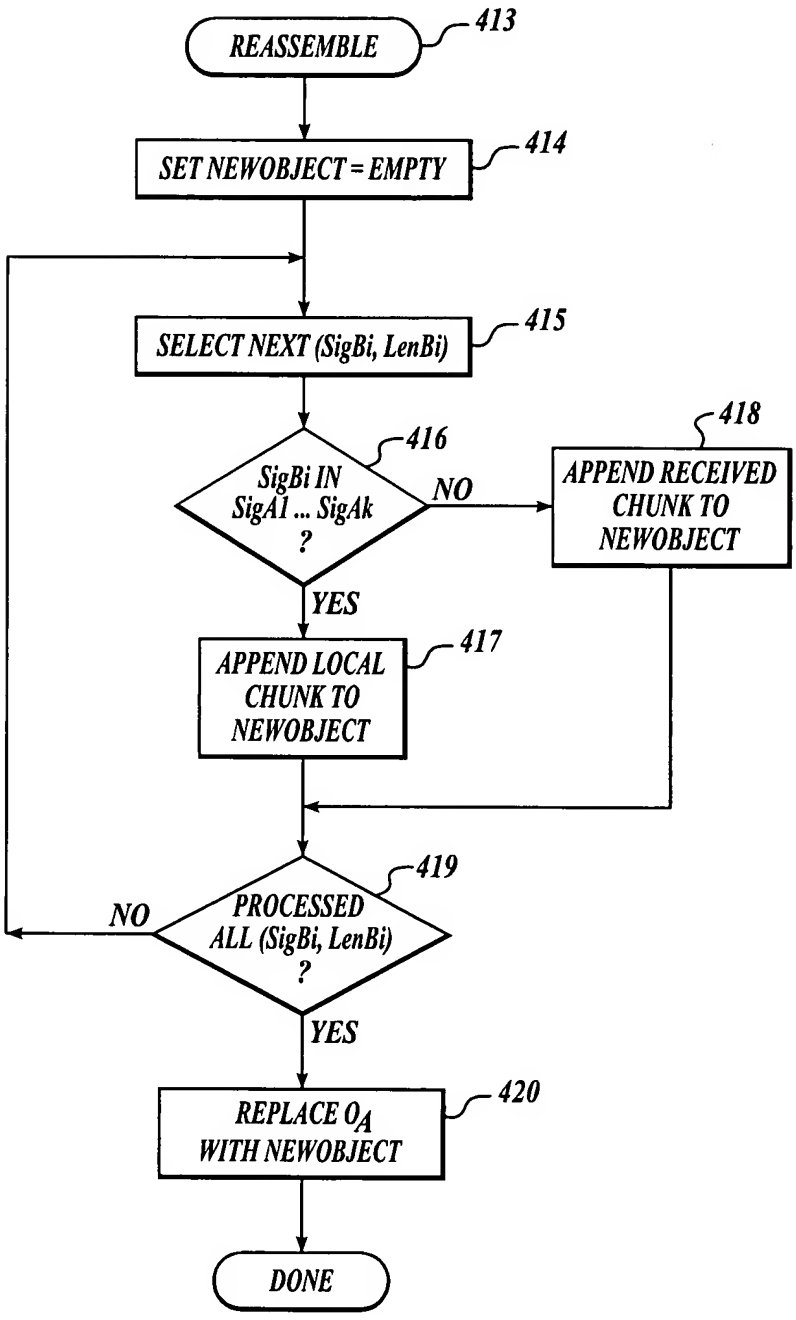
REASSEMBLE
413
SET NEWOBJECT = EMPTY
414
SELECT NEXT (SigBi, LenBi)
415
SigBi IN SigA1 ... SigAk ?
416
NO
APPEND RECEIVED CHUNK TO NEWOBJECT
418
YES
APPEND LOCAL CHUNK TO NEWOBJECT
417
PROCESSED ALL (SigBi, LenBi) ?
419
NO
YES
REPLACE O_A WITH NEWOBJECT
420
DONE

FIG. 4B

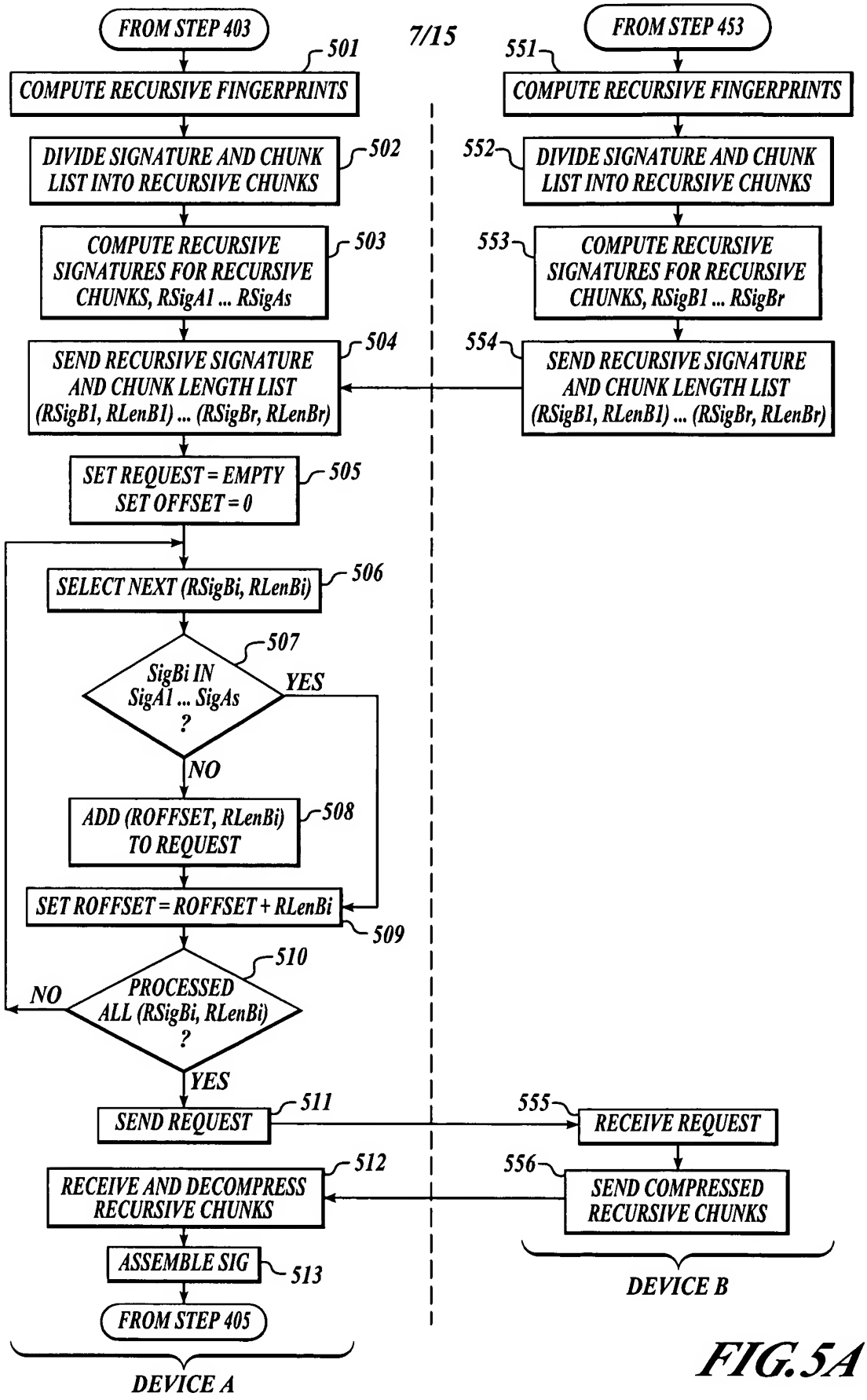
FROM STEP 403
501
COMPUTE RECURSIVE FINGERPRINTS
DIVIDE SIGNATURE AND CHUNK LIST INTO RECURSIVE CHUNKS
502
COMPUTE RECURSIVE SIGNATURES FOR RECURSIVE CHUNKS, RSigA1 ... RSigAs
503
SEND RECURSIVE SIGNATURE AND CHUNK LENGTH LIST (RSigB1, RLenB1) ... (RSigBr, RLenBr)
504
SET REQUEST = EMPTY
SET OFFSET = 0
505
SELECT NEXT (RSigBi, RLenBi)
506
SigBi IN SigA1 ... SigAs ?
507
YES
NO
ADD (ROFFSET, RLenBi) TO REQUEST
508
SET ROFFSET = ROFFSET + RLenBi
509
PROCESSED ALL (RSigBi, RLenBi) ?
510
NO
YES
SEND REQUEST
511
RECEIVE AND DECOMPRESS RECURSIVE CHUNKS
512
ASSEMBLE SIG
513
FROM STEP 405
DEVICE A
FROM STEP 453
551
COMPUTE RECURSIVE FINGERPRINTS
552
DIVIDE SIGNATURE AND CHUNK LIST INTO RECURSIVE CHUNKS
553
COMPUTE RECURSIVE SIGNATURES FOR RECURSIVE CHUNKS, RSigB1 ... RSigBr
554
SEND RECURSIVE SIGNATURE AND CHUNK LENGTH LIST (RSigB1, RLenB1) ... (RSigBr, RLenBr)
555
RECEIVE REQUEST
556
SEND COMPRESSED RECURSIVE CHUNKS
DEVICE B

FIG.5A

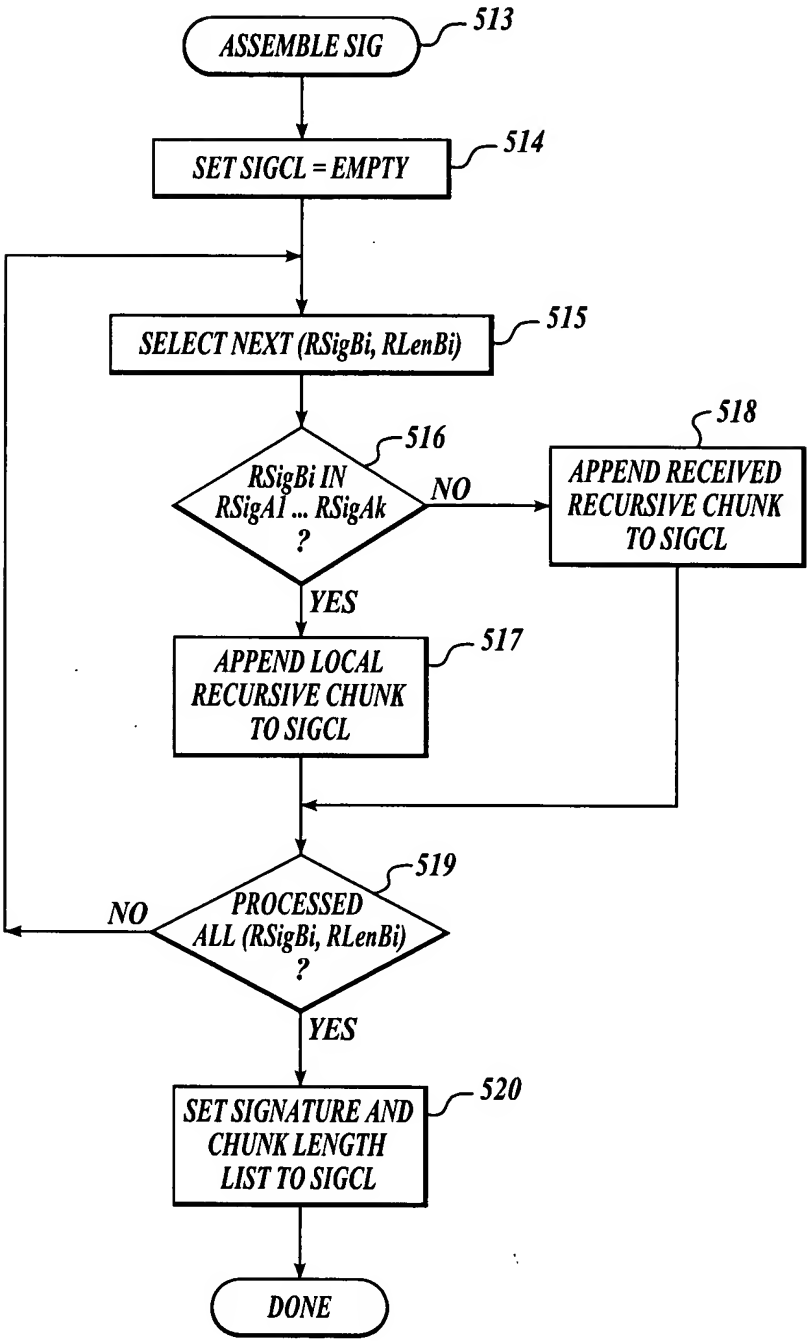
ASSEMBLE SIG
513
SET SIGCL = EMPTY
514
SELECT NEXT (RSigBi, RLenBi)
515
RSigBi IN RSigA1 ... RSigAk ?
516
NO
APPEND RECEIVED RECURSIVE CHUNK TO SIGCL
518
YES
APPEND LOCAL RECURSIVE CHUNK TO SIGCL
517
PROCESSED ALL (RSigBi, RLenBi) ?
519
NO
YES
SET SIGNATURE AND CHUNK LENGTH LIST TO SIGCL
520
DONE

FIG.5B

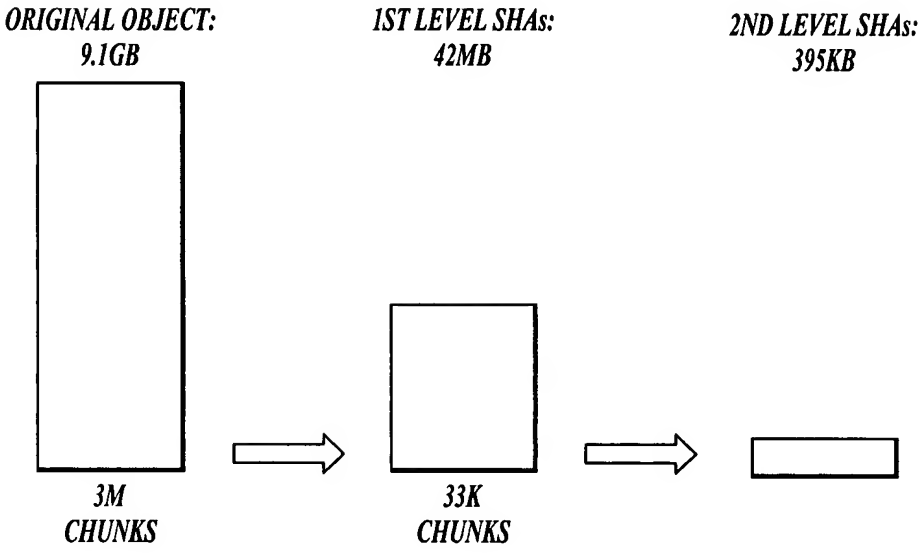
ORIGINAL OBJECT:
9.1GB
1ST LEVEL SHAs:
42MB
2ND LEVEL SHAs:
395KB
3M
CHUNKS
33K
CHUNKS

FIG. 6

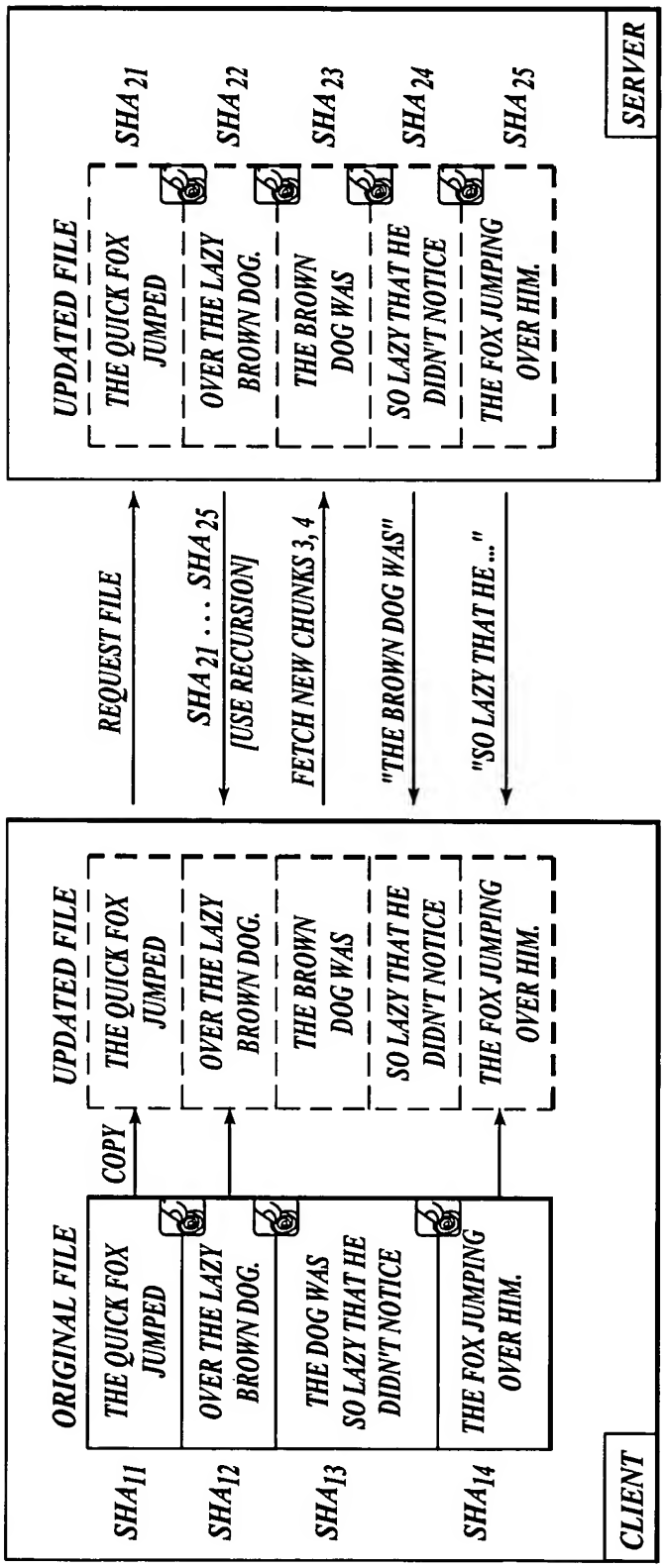
CLIENT
ORIGINAL FILE
THE QUICK FOX JUMPED
OVER THE LAZY BROWN DOG.
THE DOG WAS SO LAZY THAT HE DIDN'T NOTICE
THE FOX JUMPING OVER HIM.
SHA11
SHA12
SHA13
SHA14
COPY
UPDATED FILE
REQUEST FILE
SHA21 . . . SHA25 [USE RECURSION]
FETCH NEW CHUNKS 3, 4
"THE BROWN DOG WAS"
"SO LAZY THAT HE ..."
SERVER
UPDATED FILE
THE QUICK FOX JUMPED
OVER THE LAZY BROWN DOG.
THE BROWN DOG WAS
SO LAZY THAT HE DIDN'T NOTICE
THE FOX JUMPING OVER HIM.
SHA21
SHA22
SHA23
SHA24
SHA25

FIG. 7

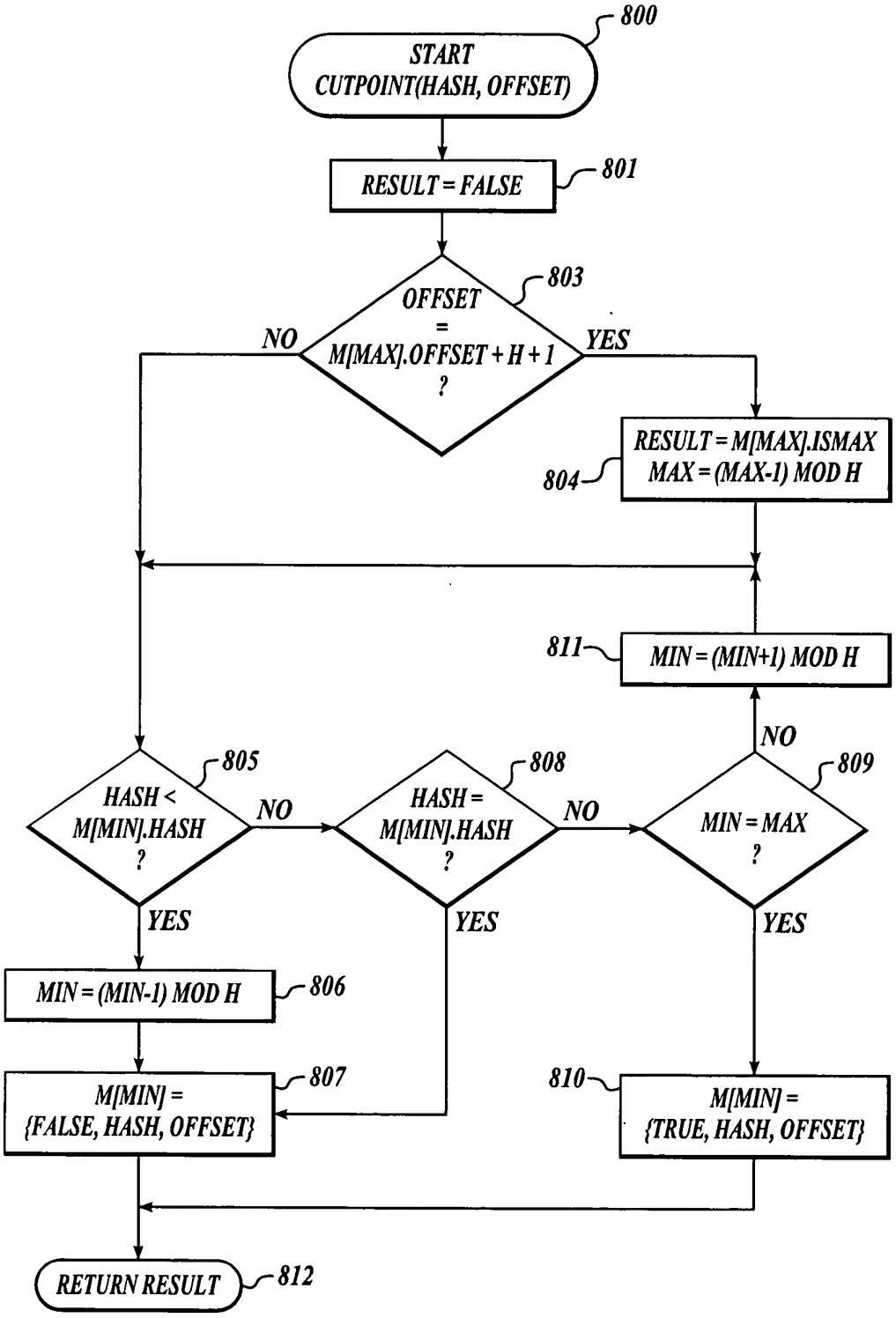
800
START
CUTPOINT(HASH, OFFSET)
801
RESULT = FALSE
803
OFFSET = M[MAX].OFFSET + H + 1 ?
NO
YES
804
RESULT = M[MAX].ISMAX
MAX = (MAX-1) MOD H
811
MIN = (MIN+1) MOD H
NO
805
HASH < M[MIN].HASH ?
NO
808
HASH = M[MIN].HASH ?
NO
809
MIN = MAX ?
YES
YES
YES
806
MIN = (MIN-1) MOD H
807
M[MIN] = {FALSE, HASH, OFFSET}
810
M[MIN] = {TRUE, HASH, OFFSET}
812
RETURN RESULT

FIG.8

```
structure Entry
  var isMax  as Boolean = false
  var hash   as Integer = 0
  var offset as Integer = 0

class LocalMaxCut
  h as Integer
  var min as Integer = 0
  var max as Integer = 0
  var M as Array of entry = new Entry[h]

  CutPoint(hash as Integer, offset as Integer) as Boolean
    var result = false
    step
      if M[max].offset + h + 1 = offset then
        result := M[max].isMax
        max := (max+1) mod h
    step
     while true do step
          step
           if M[min].hash > hash then
                 step
                   min := (min-1) mod h
                 step
                   M[min] := Entry(false, hash, offset)
                   return result
           if M[min].hash = hash then
                 M[min] := Entry(false, hash, offset)
                 return result
           if M[min].hash < hash and min = max then
                 M[min] := Entry(true, hash, offset)
                 return result
           step
                 min := (min+1) mod h
```

FIG.9

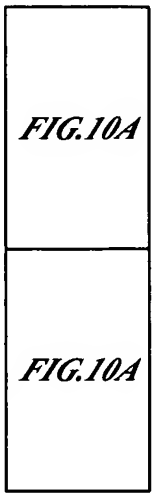

FIG.10

```
structure Entry
  var offset as Integer = 0
  var isMax  as Boolean = false
  var hash   as Integer = 0

class LocalMaxCut
  horizon as Integer
  var hashes as Seq of Integer
  var k as Integer = 0
  var l as Integer = 0
  var A as Array of Entry = new Entry[horizon]
  var B as Array of Entry = new Entry[horizon]

  CutPoints() as Seq of Integer
    var cuts as Seq of Integer = []
    for window = 0 to Length(hashes)/horizon do step
      let first = window*horizon
      let last = min((window+1)*horizon,Length(hashes)))-1
      cuts := cuts + CutPoint(first, last)
    return cuts
```

FIG.10A

```
CutPoint(first as Integer, last as Integer) as Seq of Integer
 step // Initialize A with the first entry at the offset
  k := 0
  A[0] := Entry(last,true,hashes[last])
  last := last - 1
 step // Update A[k] in the interval up to B[l]'s horizon
  while last > B[l].offset + horizon do step
   Insert(last)
   last := last - 1
 step // Update A[k] and B[l] in the remaining interval
  while last >= first do step
   Insert(last)
   if B[l].hash <= hashes[last] then
     B[l].isMax := false
   last := last -1
 step // determine whether A[k] is a cutpoint with respect to B
  A[k].isMax := A[k].isMax and
            forall j in 0..l holds
            (B[j].offset + horizon < A[k].offset or
             B[j].hash < A[k].hash)
 step // Set B to A for the next round and return cut-point
  B, l := A, k
  return if B[l].isMax then [B[l].offset] else []
```

FIG.10B

```
class LocalMaxCut
  Insert(offset as Integer)
    if hashes[offset] >= A[k].hash then
      if hashes[offset] = A[k].hash then
        // duplicated hashes within distance
        // of "horizon" are not maximal.
        A[k].isMax := false
      else
        A[k+1] := Entry(offset, true, hashes[offset])
        k := k + 1
```

FIG.11